Sibling-First Recursive Graph Drawing for Visualizing Java Bytecode

Md. Jawaherul Alam, Michael T. Goodrich, and Timothy Johnson

Dept. of Computer Science, University of California, Irvine, CA USA
{alamm1,goodrich,tujohnso@uci.edu}@uci.edu

1 Introduction

We describe a tool, the JVM abstracting abstract machine (Jaam) Visualizer, or "J-Viz" for short, which is intended for use by security analysts to perform such searches through the exploration of graphs derived from Java bytecode. The workflow for our tool involves taking a given program, specified in Java bytecode, and constructing a control-flow graph of the possible execution paths for this software, using a framework known as *control flow analysis* (CFA) [6]. Our tool then provides a human analyst with an interactive view of this graph, including heuristics for aiding the identification of the suspicious parts.

One of the main components of our J-Viz tool involves visualizing control-flow graphs in a canonical way based on a novel vertex numbering scheme that we call the *sibling-first recursive* numbering. This numbering scheme is essentially a hybrid between the well-known breadth-first and depth-first numbering schemes, but differs from both in a way that appears to be more useful for visualizing control-flow graphs. In particular, this tends to highlight areas in software where code is repeated and allows us to provide visual highlights of code that is contained in deeply nested loops. We had the following goals in mind:

- We want users to recognize patterns in source code from our visualization; similar code sections should produce similar subgraphs, drawn similar way.
- We want to use a hierarchical visualization, in which users can collapse or expand sections of the graph to different levels of detail. But we also want to preserve a consistent mental model of the graph. Thus, drawings should not drastically shift the vertex positions when sections are collapsed or expanded.
- No matter what sequence of actions the user performs, drawings should be consistent. That is, the same view of a graph, in which the same set of nodes are collapsed and expanded, should always be drawn in the same way.
- Our system should rank sections of the graph by how likely they are to produce vulnerabilities, and display this information visually to the user.

We believe that J-Viz makes substantial progress in achieving these goals, and we provide several case studies in this paper that support this conclusion. Please see the full version of the paper [1] for more details.

Related Work and Main Contributions. Visualization tools have also previously been applied to source code. Doxygen [5], a tool for automatically generating documentation, can produce various kinds of diagrams for visualizing code, including call graphs. It is generally configured to use the *dot* [4] tool from GraphViz to draw these graphs hierarchically. Similarly, Visual Studio can visualize call graphs to aid programmers in debugging applications [3]. In constrast with these systems, our J-Viz tool provides four main features that these tools do not provide. First, J-Viz shows a greater level of detail, since it analyzes code at the level of individual instructions rather than methods. Second, J-Viz allows the user to interact with a graph and produce multiple views of the same Java bytecode. Third, the layout algorithm used in J-Viz is designed to draw similar code fragments in the same (canonical) way, so as to highlight portions of repeated code. Fourth, J-Viz guides the user to potential security vulnerabilities, by highlighting nodes that are believed to be risky based on algorithmic complexity (or other factors), whereas these other systems were not focused on software security. Another tool, Jinsight [2], can be used to profile a Java program to provide various views of resource usage, such as highlighting which instances of a class have taken the most time or used the most memory. This tool does not provide a full graph of the program's possible execution paths, however, which we believe to be essential for detecting security vulnerabilities.

Acknowledgements. This reports on work supported by DARPA under agreement AFRL FA8750-15-2-0092. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. This work was also supported NSF grants 1228639 and 1526631. We also thank David Eppstein, Matthew Might, William Byrd, Michael Adams, and Guannan Wei for helpful discussions.

References

1. Alam, M.J., Goodrich, M.T., Johnson, T.: J-Viz: Sibling-first recursive graph drawing for visualizing java bytecode. CoRR abs/1608.08970 (2016)
2. De Pauw, W., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J., Yang, J.: Visualizing the execution of java programs. In: Software Visualization, pp. 151–162. Springer (2002)
3. DeLine, R., Venolia, G., Rowan, K.: Software development with code maps. Commun. ACM 53(8), 48–54 (Aug 2010)
4. Gansner, E.R., Koutsofios, E., North, S.C., Vo, G.P.: A technique for drawing directed graphs. Software Engineering, IEEE Transactions on 19(3), 214–230 (1993)
5. van Heesch, D.: Doxygen: Source code documentation generator tool. Available online: http://www.doxygen.org (accessed on 8 June 2016) (2008)
6. Shivers, O.G.: Control-Flow Analysis of Higher-Order Languages. Ph.D. thesis, Carnegie Mellon University (1991)